Embedded Labworks

EMBEDDED LINUX CONFERENCE 2022

DESIGNING SECURE CONTAINERIZED APPLICATIONS FOR EMBEDDED LINUX DEVICES

Sergio Prado, Embedded Labworks
https://www.linkedin.com/in/sprado

WHOAMI

- Designing and developing embedded software for 25+ years (Embedded Linux, Embedded Android, RTOS, etc).
- Consultant and trainer at Embedded Labworks for 10+ years.
  https://e-labworks.com/en
- Open source software contributor (Buildroot, Yocto Project, Linux kernel, etc).
- Blogger at EmbeddedBits.org.
  https://embeddedbits.org/

Embedded Labworks

AGENDA

- Overview of container technology and the usage of containers on embedded systems.
- Introduction to container security.
- Securing container images:
  - *Built-time security*: creating more secure container images.
  - *Run-time security*: securing the execution of the container.

DISCLAIMERS

- I am just a developer (worried about the security), but I am not a security expert!
- Security requires a holistic approach (defense in depth).
  - In this talk, we will just cover a small part of how to secure an embedded design based on containers.
- We will use Docker in the examples, but the concepts apply to other container runtimes and tools as well.

WHAT IS A CONTAINER?

- It's just a convenient way to distribute and execute software!
  - It can bundle a small application binary with its dependencies as a self-contained executable image.
  - It can contain a complete Linux system.
- Containers run isolated from the host with the help of a few kernel features, including:
  - *Namespaces* isolate kernel resources (pid, uts, user, mnt, net, ipc, etc).
  - *Switch root* makes it possible to change the location of the root filesystem.
  - *Control groups* limit resource usage (memory, CPU, etc).

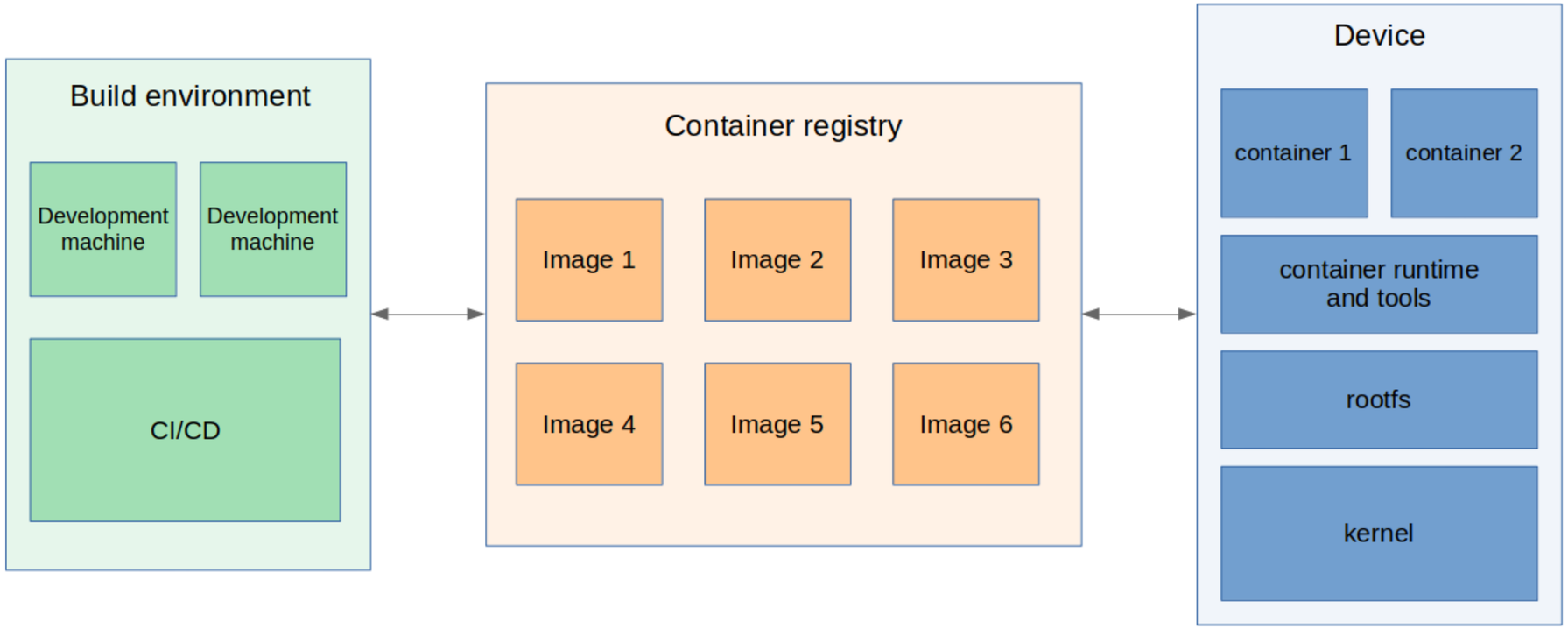
WHY CONTAINERS ON EMBEDDED?

- *Productivity*: focus on application development (less time spent on creating and maintaining an infrastructure to run the application).
- *Isolation*: Make it possible to have different execution environments in the same host.
- *Modularity*: encourage the development of a more modular system, improving maintenance, portability and reuse.
- *Control*: more control over the usage of hardware and software resources.
- *Updatability*: easier to implement a more robust, fast and fail-safe update system.
- *Security*: provides an infrastructure to improve security!

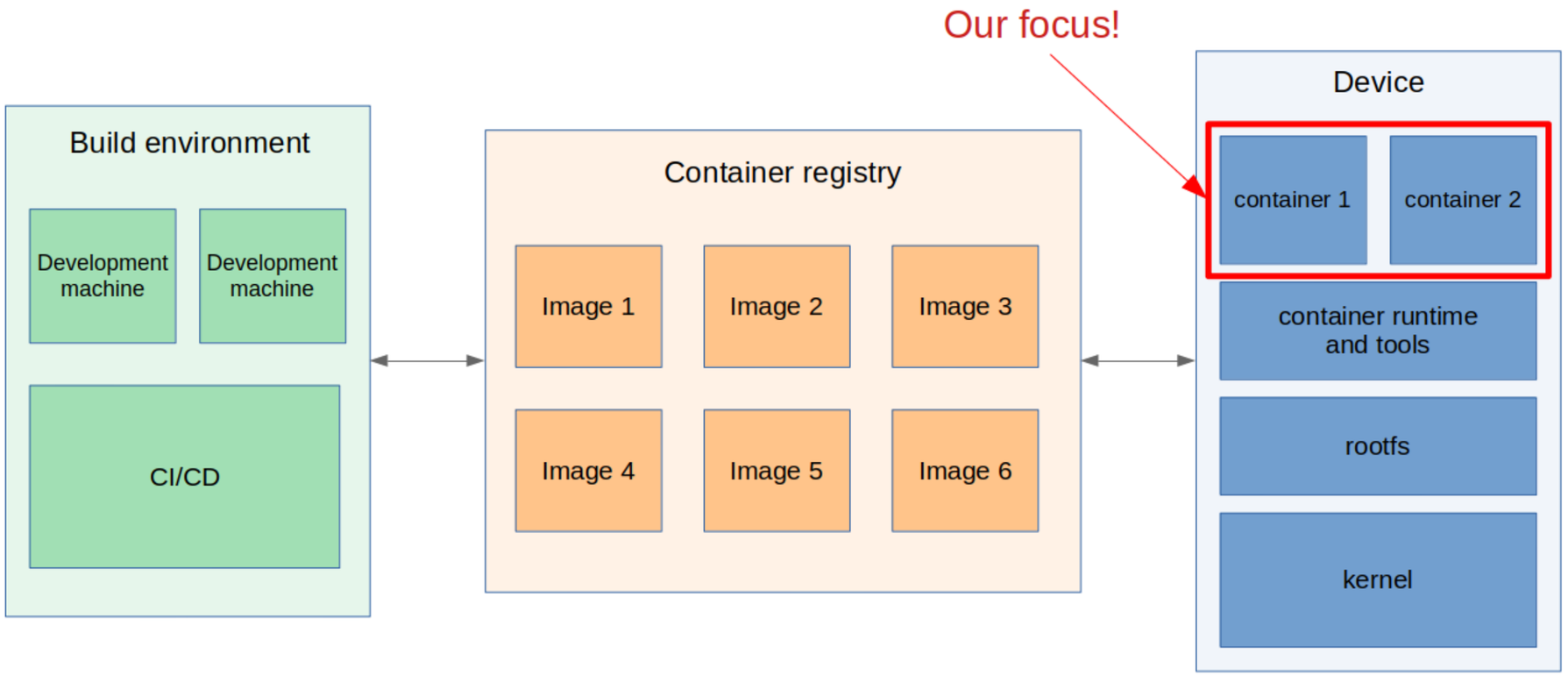
ARE CONTAINERS SECURE?

- Not by default!
  - But they provide an infrastructure that facilitates a secure design.
- With the correct design, running applications inside a container will make it much harder for an attacker to explore vulnerabilities and get access to the host OS or application's data.
- In the end, security is like an onion, with several layers. Containers are just one of those layers you might want to protect!

CONTAINER INFRASTRUCTURE



OUR FOCUS!


Our focus!
Build environment
Development machine
Development machine
CI/CD
Container registry
Image 1
Image 2
Image 3
Image 4
Image 5
Image 6
Device
container 1
container 2
container runtime and tools
rootfs
kernel

THIS TALK IS ABOUT...

- Securing a container image!
  - How to build a more secure container image?
  - How to improve the security of the container at runtime?
- When securing a container image, there are a few important security concepts we need to understand:
  - *Economy of mechanism*: keep the design as simple and small as possible.
  - *Least privilege*: containers should run with the least set of privileges necessary to complete their job.

LET'S CONTAINERIZE THIS APP!

```c
int main(void)
{
    const char rtc[] = "/dev/rtc0";
    struct rtc_time rtc_tm;
    int fd;

    if ((fd = open(rtc, O_RDONLY)) == -1) {
        perror(rtc);
        exit(errno);
    }

    if (ioctl(fd, RTC_RD_TIME, &rtc_tm) == -1) {
        perror("RTC_RD_TIME ioctl");
        exit(errno);
    }

    printf("Current RTC date/time is %04d-%02d-%02d %02d:%02d:%02d\n",
        rtc_tm.tm_year + 1900, rtc_tm.tm_mon + 1, rtc_tm.tm_mday,
        rtc_tm.tm_hour, rtc_tm.tm_min, rtc_tm.tm_sec);

    return 0;
}
```

EMBEDDED LINUX CONFERENCE 2022

PART 1: SECURING THE CONTAINER IMAGE

SECURING THE CONTAINER IMAGE

- Create a minimal container image.
- Create and run images you trust.
- Run static analysis tools.
- Run security scanning tools.
- Make it easily updatable.

CREATE A MINIMAL CONTAINER IMAGE

- The final container image should have only the required components for the application to run.
- Smaller images have fewer attack vectors, decreasing the chances of an attacker to explore a vulnerability and escalate privileges inside the container.
- A few strategies to build a minimal container image:
  - Build upon a minimal base image (Alpine, Google distroless).
  - Use multi-stage builds.
  - Create containers with statically linked applications.
  - Use a build system (Yocto Project/OpenEmbedded, Buildroot).

HANDS-ON: DEBIAN BASED IMAGE

```
# Dockerfile.debian
FROM debian:bullseye-slim

RUN apt-get update \
    && apt-get install -y gcc \
    && rm -rf /var/lib/apt/lists/*

COPY app.c .

RUN gcc app.c -o app

CMD ["./app"]
```

```
$ docker build -f Dockerfile.debian -t sergioprado/debian-app:1.0.0 .

$ docker image ls sergioprado/debian-app:1.0.0
REPOSITORY               TAG       IMAGE ID       CREATED         SIZE
sergioprado/debian-app   1.0.0     98df5fef2f51   6 seconds ago   250MB

$ docker run --rm -v /dev:/dev --privileged sergioprado/debian-app:1.0.0
Current RTC date/time is 2022-06-07 11:55:19
```

HANDS-ON: ALPINE WITH MULTI-STAGE BUILD

```
# Dockerfile.alpine
FROM alpine:3.16.0 as build

RUN apk add --no-cache gcc musl-dev linux-headers

COPY app.c .

RUN gcc app.c -o app

FROM alpine:3.16.0

COPY --from=build app .

CMD ["./app"]
```

```
$ docker build -f Dockerfile.alpine -t sergioprado/alpine-app:1.0.0 .

$ docker image ls sergioprado/alpine-app:1.0.0
REPOSITORY                TAG       IMAGE ID       CREATED          SIZE
sergioprado/alpine-app    1.0.0     2e8481138eab   8 seconds ago    5.55MB

$ docker run --rm -v /dev:/dev --privileged sergioprado/alpine-app:1.0.0
Current RTC date/time is 2022-06-07 12:00:21
```

HANDS-ON: STATICALLY LINKED BINARY

```
# Dockerfile.alpine-static
FROM alpine:3.16.0 as build

RUN apk add --no-cache gcc musl-dev linux-headers

COPY app.c .

RUN gcc app.c -o app -static

FROM scratch

COPY --from=build app .

CMD ["./app"]
```

```
$ docker build -f Dockerfile.alpine-static -t sergioprado/alpine-app-static:1.0.0 .

$ docker image ls sergioprado/alpine-app-static:1.0.0
REPOSITORY                      TAG       IMAGE ID       CREATED         SIZE
sergioprado/alpine-app-static   1.0.0     d19bb40b7605   5 seconds ago   135kB

$ docker run --rm -v /dev:/dev --privileged sergioprado/alpine-app-static:1.0.0
Current RTC date/time is 2022-06-07 12:03:34
```

CREATE AND RUN IMAGES YOU TRUST

- Use base images from trusted sources.
- Use the digest of base images (instead of a tag) when building container images.
  - Tags are a dynamic reference to an image version at a specific point in time.
  - Digests are immutable and deterministic, and so more secure!
- A public-key cryptography scheme can be used to guarantee the authenticity of the container image.
  - Two options to check the authenticity of a container image are Docker Content Trust (DCT) and cosign.

HANDS-ON: USING DIGESTS TO CREATE AN IMAGE

```
# Dockerfile.alpine-static
FROM alpine@sha256:4ff3ca91275773af45cb4b0834e12b7eb47d1c18f770a0b151381cd227f4c253 as build

RUN apk add --no-cache gcc musl-dev linux-headers

COPY app.c .

RUN gcc app.c -o app -static

FROM scratch

COPY --from=build app .

CMD ["./app"]
```

```
$ docker build -f Dockerfile.alpine-static -t sergioprado/alpine-app-static:1.0.0 .

$ docker image ls sergioprado/alpine-app-static:1.0.0
REPOSITORY                      TAG       IMAGE ID       CREATED              SIZE
sergioprado/alpine-app-static   1.0.0     d19bb40b7605   About a minute ago   135kB

$ docker run --rm -v /dev:/dev --privileged sergioprado/alpine-app-static:1.0.0
Current RTC date/time is 2022-06-07 12:03:34
```

HANDS-ON: USING DIGESTS TO RUN CONTAINERS

```
$ docker image ls --digests sergioprado/alpine-app-static:1.0.0
REPOSITORY                      TAG       DIGEST    IMAGE ID       CREATED         SIZE
sergioprado/alpine-app-static   1.0.0     <none>    d19bb40b7605   2 minutes ago   135kB

$ docker push sergioprado/alpine-app-static:1.0.0
The push refers to repository [docker.io/sergioprado/alpine-app-static]
d0cc8476ff2c: Pushed
1.0.0: digest: sha256:2554754efe58c1c81ea267fcbba68cceeb0d8d64ea9488a2ffc00b465153747 size: 526

$ docker images --digests sergioprado/alpine-app-static
REPOSITORY                      TAG       DIGEST                    IMAGE ID       CREATED
sergioprado/alpine-app-static   1.0.0     sha256:2554754efe58...    d19bb40b7605   3 minutes ago

$ docker run --rm -v /dev:/dev --privileged sergioprado/alpine-app-static@sha256:2554754efe5...
Current RTC date/time is 2022-06-07 13:54:26
```

HANDS-ON: CREATING SIGNING KEYS

```
$ docker trust key generate app-key
Generating key for app-key...
Enter passphrase for new app-key key with ID 4edf2b8:
Repeat passphrase for new app-key key with ID 4edf2b8:
Successfully generated and loaded private key.
Corresponding public key available: /home/sprado/Temp/talk/app-key.pub

$ ls app-key.pub
app-key.pub

$ ls ~/.docker/trust/private/
4edf2b82059f6eb2cad93b9033a53e5e4832752d25849235a6cef3c40aa30f31.key

$ docker trust signer add app-key --key app-key.pub sergioprado/alpine-app-static
```

HANDS-ON: RUNNING SIGNED IMAGES

```
$ docker image ls sergioprado/alpine-app-static
REPOSITORY                      TAG       IMAGE ID       CREATED         SIZE
sergioprado/alpine-app-static   1.0.0     d19bb40b7605   6 minutes ago   135kB

$ docker image rm -f d19bb40b7605

$ export DOCKER_CONTENT_TRUST=1

$ docker run --rm -v /dev:/dev --privileged sergioprado/alpine-app-static:1.0.0
docker: No valid trust data for 1.0.0.

$ docker build -f Dockerfile.alpine-static -t sergioprado/alpine-app-static:1.0.0 .

$ docker push sergioprado/alpine-app-static:1.0.0
The push refers to repository [docker.io/sergioprado/alpine-app-static]
d0cc8476ff2c: Layer already exists
1.0.0: digest: sha256:42b51ac5882f4696de15715650d028b7e711a09892170c01a9ee8ae416018458 size: 526
Signing and pushing trust metadata
Enter passphrase for app-key key with ID 4edf2b8:
Successfully signed docker.io/sergioprado/alpine-app-static:1.0.0

$ docker run --rm -v /dev:/dev --privileged sergioprado/alpine-app-static:1.0.0
Current RTC date/time is 2022-06-07 14:18:03
```

STATIC ANALYSIS TOOLS

- Regardless of whether or not you are working with containers, always use static analysis tools!
- Static analysis tools are able to analyze the source code (without running the program) to find problems before they happen.
- These tools can find program errors like null pointer dereferences, memory leaks, integer overflow, out-of-bounds access, use before initialization, etc!
- There are many good open-source (*cppcheck*, *splint*, *clang*, etc) and commercial (*Coverity*, *PC-Lint*, etc) options for static code analysis.
  - Nowadays the compiler is a very good static analysis tool - just enable the warnings!
  - Integrate it in a CI environment so you can catch errors as soon as possible!

HANDS-ON: USING CPPCHECK

```
# Dockerfile.alpine-static
FROM alpine@sha256:4ff3ca91275773af45cb4b0834e12b7eb47d1c18f770a0b151381cd227f4c253 as build

RUN apk add --no-cache gcc musl-dev linux-headers cppcheck

COPY app.c .

RUN cppcheck --error-exitcode=1 app.c

RUN gcc app.c -o app -static -Werror -Wall

FROM scratch

COPY --from=build app .

CMD ["./app"]
```

```
$ docker build -f Dockerfile.alpine-static -t sergioprado/alpine-app-static:1.0.0 .
Step 1/8 : FROM alpine@sha256:4ff3ca91275773af45cb4b0834e12b7eb47d1c18f770a0b151381... as build
Step 2/8 : RUN apk add --no-cache gcc musl-dev linux-headers cppcheck
Step 3/8 : COPY app.c .
Step 4/8 : RUN cppcheck --error-exitcode=1 app.c
Checking app.c ...
app.c:17:7: error: Uninitialized variable: fd [uninitvar]
 if ((fd == open(rtc, O_RDONLY)) == -1) {
      ^
The command '/bin/sh -c cppcheck --error-exitcode=1 app.c' returned a non-zero code: 1
```

SECURITY SCANNING

- Security scanning tools are able to find known security vulnerabilities in containers.
  - Depending on the tool, the analysis is done at different levels: source-code (e.g. Dockerfile), binary (image layers) and at runtime.
  - Might not be needed if you have only a static linked application inside the container image.
- There are a few open-source tools available, including *Trivy* (from Aqua Security), *Grype* (from Anchore) and *Clair* (from CoreOS).
  - There is also a scanning tool integrated into Docker (*docker scan*), but it is a paid service (10 free scans per month).
  - A good approach is to integrate those tools into a CI/CD environment.

HANDS-ON: SCANNING WITH TRIVY

```
$ trivy image sergioprado/debian-app:1.0.0
2022-06-09T14:04:49.927-0300    INFO    Detected OS: debian
2022-06-09T14:04:49.927-0300    INFO    Detecting Debian vulnerabilities...
2022-06-09T14:04:49.947-0300    INFO    Number of language-specific files: 0

sergioprado/debian-app:1.0.0 (debian 11.3)

Total: 317 (UNKNOWN: 2, LOW: 233, MEDIUM: 43, HIGH: 35, CRITICAL: 4)
...

$ trivy image sergioprado/alpine-app:1.0.0
2022-06-09T14:05:45.671-0300    INFO    Detected OS: alpine
2022-06-09T14:05:45.671-0300    INFO    This OS version is not on the EOL list: alpine 3.16
2022-06-09T14:05:45.671-0300    INFO    Detecting Alpine vulnerabilities...
2022-06-09T14:05:45.671-0300    INFO    Number of language-specific files: 0

sergioprado/alpine-app:1.0.0 (alpine 3.16.0)

Total: 0 (UNKNOWN: 0, LOW: 0, MEDIUM: 0, HIGH: 0, CRITICAL: 0)

$ trivy image sergioprado/alpine-app-static:1.0.0
2022-06-09T14:06:03.849-0300    INFO    Number of language-specific files: 0
```

Embedded Labworks

HANDS-ON: TRIVY OUTPUT

sprado@sprado-office: ~

```
sprado@sprado-office:~$ trivy image sergioprado/app-debian:1.0.0
2022-06-07T15:41:29.179-0300    INFO    Detected OS: debian
2022-06-07T15:41:29.179-0300    INFO    Detecting Debian vulnerabilities...
2022-06-07T15:41:29.199-0300    INFO    Number of language-specific files: 0

sergioprado/app-debian:1.0.0 (debian 11.3)

Total: 316 (UNKNOWN: 1, LOW: 233, MEDIUM: 42, HIGH: 36, CRITICAL: 4)
```

Library	Vulnerability	Severity	Installed Version	Fixed Version	Title
apt	CVE-2011-3374	LOW	2.2.4		It was found that apt-key in apt, all versions, do not correctly... https://avd.aquasec.com/nvd/cve-2011-3374
binutils	CVE-2017-13716		2.35.2-2		binutils: Memory leak with the C++ symbol demangler routine in libiberty https://avd.aquasec.com/nvd/cve-2017-13716
	CVE-2018-12934				binutils: Uncontrolled Resource Consumption in remember_Ktype in cplus-dem.c https://avd.aquasec.com/nvd/cve-2018-12934
	CVE-2018-18483				binutils: Integer overflow in cplus-dem.c:get_count() allows for denial of service https://avd.aquasec.com/nvd/cve-2018-18483
binutils	CVE-2018-20623	LOW	2.35.2-2		binutils: Use-after-free in the error function https://avd.aquasec.com/nvd/cve-2018-20623
	CVE-2018-20673				libiberty: Integer overflow in demangle_template() function https://avd.aquasec.com/nvd/cve-2018-20673
	CVE-2018-20712				libiberty: heap-based buffer over-read in d_expression_1 https://avd.aquasec.com/nvd/cve-2018-20712
binutils	CVE-2018-9996	LOW	2.35.2-2		binutils: Stack-overflow in libiberty/cplus-dem.c causes crash https://avd.aquasec.com/nvd/cve-2018-9996
binutils	CVE-2019-1010204	LOW	2.35.2-2		binutils: Improper Input Validation, Signed/Unsigned Comparison, Out-of-bounds Read in gold/fileread.cc and elfcpp/elfcpp_file.h... https://avd.aquasec.com/nvd/cve-2019-1010204
binutils	CVE-2020-35448	LOW	2.35.2-2		binutils: Heap-based buffer overflow in bfd_getl_signed_32() in libbfd.c because sh_entsize is not...

HANDS-ON: SCANNING WITH GRYPE

```
$ grype sergioprado/debian-app:1.0.0
 ✔ Loaded image
 ✔ Cataloged packages      [151 packages]
 ✔ Scanned image           [319 vulnerabilities]
NAME                       INSTALLED          FIXED-IN     TYPE  VULNERABILITY     SEVERITY
apt                        2.2.4                           deb   CVE-2011-3374     Negligible
binutils                   2.35.2-2                        deb   CVE-2018-20673    Negligible
binutils                   2.35.2-2                        deb   CVE-2018-12934    Negligible
...

$ grype sergioprado/alpine-app:1.0.0
 ✔ Loaded image
 ✔ Cataloged packages      [14 packages]
 ✔ Scanned image           [0 vulnerabilities]
No vulnerabilities found

$ grype sergioprado/alpine-app-static:1.0.0
 ✔ Loaded image
 ✔ Cataloged packages      [0 packages]
 ✔ Scanned image           [0 vulnerabilities]

No vulnerabilities found
```

EASILY UPDATABLE

- In the end, any software has bugs, and if security matters to you, having an update system in place is mandatory.
- There are different approaches to solve this problem:
  - There are simple tools that periodically monitor and trigger container updates (Watchtower, Ouroboros, etc).
  - There are complete container-based operating systems with an infrastructure for remote/automatic updates (TorizonCore, Linux microPlatform, CoreOS, etc).
- From the container perspective, the reliability and robustness of update systems might be improved by designing immutable container images (an immutable image is an image that contains everything it needs to run the application).

EMBEDDED LINUX CONFERENCE 2022

PART 2: SECURING THE CONTAINER EXECUTION

SECURING THE CONTAINER EXECUTION

- Restrict container privileges.
- Restrict syscalls.
- Control resource usage.
- Enable a security module.
- Secure network connections.
- Secure storage.

RESTRICT CONTAINER PRIVILEGES

- So far, we have been running the container with the following command:

```
$ docker run --rm -v /dev:/dev --privileged sergioprado/alpine-app-static:1.0.0
```

- This command has several issues!
  - It bind mounts the complete */dev* directory (*-v /dev:/dev*), but the container application just needs */dev/rtc0*.
  - It runs with the *root* user, with more privileges than what is needed to do its job.
  - It runs in privileged mode (*--privileged*).

HANDS-ON: WHERE IS THE SECURITY HERE?

```
$ docker run --rm -v /dev:/dev --privileged -it --entrypoint /bin/sh \
             sergioprado/alpine-app:1.0.0

# id
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),
11(floppy),20(dialout),26(tape),27(video)

# ls /dev
autofs           loop23           sdb7             tty44            ttyS9
block            loop24           sdc              tty45            ttyUSB0
bsg              loop25           sdc1             tty46            ttyUSB1
btrfs-control    loop26           sdd              tty47            ttyUSB2
bus              loop27           sdd1             tty48            ttyUSB3
char             loop28           serial           tty49            ttyUSB4
...

# mount /dev/sdb5 /mnt

# ls /mnt
bin             etc             lib32           mnt             sbin            tmp
boot            home            lib64           opt             snap            usr
cdrom           initrd.img      libx32          proc            srv             var
...
```

WHY --PRIVILEGED IS REALLY BAD?

- The Docker documentation says that the *--privileged* option "gives extended privileges to the container".
- In practice, this option will:
  - Enable all capabilities.
  - Enable access to all device files.
  - Configure AppArmor or SELinux (if enabled) to allow the container nearly all the same access to the host as processes running outside containers on the host.
- In other words, the container will be able to do almost everything that the host can!
  - This option was created to allow special use-cases, like running Docker within Docker, but it is sometimes misused.

DEVICE FILES INSIDE THE CONTAINER

- A container is not allowed to access any device files by default.
  - This is because Docker restricts access to device files using a kernel feature called cgroups devices.
- To enable access to a device file inside a container, the *--device* option can be used.
- This option will instruct Docker to:
  - Bind mount the device file inside the container.
  - Create a rule to add the device into the cgroup allowed devices list.
- Cgroups device rules can also be manually defined with the *--device-cgroup-rule* option.

HANDS-ON: ACCESSING DEVICES FILES

```
$ docker run --rm --device /dev/rtc0:/dev/rtc0 -it --entrypoint /bin/sh \
             sergioprado/alpine-app:1.0.0

# ls /dev
console  fd       mqueue   ptmx     random   shm      stdin    tty      zero
core     full     null     pts      rtc0     stderr   stdout   urandom

$ docker run --rm --device /dev/rtc0:/dev/rtc0 sergioprado/alpine-app-static:1.0.0
Current RTC date/time is 2022-06-08 12:00:42
```

USERS INSIDE CONTAINERS

- It is recommended to not run container images as root (when possible).
- In Docker, the default user (and groups) can be changed:
  - Using the *USER* instruction inside the Dockerfile.
  - Passing the *--user* option when starting a container.
- It's recommended to use randomized UIDs that don’t map to real users in the host or use the user namespace feature (covered in the following slides).
- To prevent a "normal" user inside the container from gaining new privileges during execution, for example running programs with the *setuid*/*setgid* bit set, we can use the *--security-opt no-new-privileges* option in Docker.

HANDS-ON: CHANGING USER AND GROUP

```
$ docker run --rm --device /dev/rtc0:/dev/rtc0 -it --entrypoint /bin/sh \
             sergioprado/alpine-app:1.0.0

# id
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy

./app
Current RTC date/time is 2022-06-08 12:21:46

$ docker run --rm --device /dev/rtc0:/dev/rtc0 --user 2000:2000 -it --entrypoint /bin/sh \
             sergioprado/alpine-app:1.0.0

$ id
uid=2000 gid=2000

$ ping 8.8.8.8
ping: permission denied (are you root?)

$ ./app
/dev/rtc0: Permission denied

$ ls -l /dev/rtc0
crw-------    1 root     root      249,   0 Jun  8 12:22 /dev/rtc0
```

USER NAMESPACE

- User namespaces isolate security-related identifiers and attributes (UIDs, GIDs, etc).
- The best way to prevent privilege-escalation attacks from within a container is to configure the container to run with unprivileged users.
- For containers whose processes must run as the *root* user within the container, you can re-map this user to a less-privileged user on the Docker host.
  - The user and group IDs mappings should be configured in the host OS (*/etc/subuid* and */etc/subgid*).
  - The Docker daemon should be started with the *--userns-remap* option.
- To disable user namespaces for a specific container, use the *--userns=host* option.

HANDS-ON: ENABLING USER NAMESPACE

```
$ cat /etc/subuid
sprado:100000:65536

$ cat /etc/subgid
sprado:100000:65536

$ cat /etc/docker/daemon.json
{
    "max-concurrent-uploads": 1,
    "max-concurrent-downloads": 3,
    "userns-remap": "sprado"
}

$ docker info 2>&- | grep -E Root\|userns
  userns
 Docker Root Dir: /var/lib/docker/100000.100000
```

HANDS-ON: USER NAMESPACE IN CONTAINERS

```
$ docker run --rm --device /dev/rtc0:/dev/rtc0 -it --entrypoint /bin/sh \
             sergioprado/alpine-app:1.0.0

# id
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),
11(floppy),20(dialout),26(tape),27(video)

# ping 8.8.8.8 -c 1
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=117 time=11.898 ms
1 packets transmitted, 1 packets received, 0% packet loss

# ./app
/dev/rtc0: Permission denied

$ ls -l /dev/rtc0
crw-------    1 root     root      249,   0 Jun  8 12:22 /dev/rtc0

$ docker run --rm --device /dev/rtc0:/dev/rtc0 --userns host sergioprado/alpine-app-static:1.0.0
Current RTC date/time is 2022-06-08 13:26:42
```

CAPABILITIES

- To check for permissions, traditional UNIX implementations distinguish two categories of processes:
  - Privileged processes: those running with effective user ID 0 (superuser, root).
  - Unprivileged processes: those running with a nonzero effective UID.
- Privileged processes bypass all kernel permission checks.
- Unprivileged processes are subject to full permission checking based on the process's credentials.

CAPABILITIES (CONT.)

- Since Linux version 2.2, Linux divides the privileges traditionally associated with superuser into distinct units known as capabilities, which can be independently enabled/disabled (*CAP_NET_ADMIN*, *CAP_NET_RAW*, *CAP_SYS_ADMIN*, etc).
- In Docker, the capabilities of a container running as *root* can be configured through the *--cap-add* and *--cap-drop* options.
- As a security measure, we should always drop all capabilities of a container running as *root*, and enable just those needed for the container application to do its job:

```
$ docker run --cap-drop all --cap-add NET_ADMIN ...
```

- The Tracee tool can be used to trace container execution and identify the required capabilities.

HANDS-ON: CAPABILITIES

```
$ docker run --rm --device /dev/rtc0:/dev/rtc0 -it --entrypoint /bin/sh \
             sergioprado/alpine-app:1.0.0

# getpcaps 1
1: cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,
cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap=eip

$ docker run --rm --device /dev/rtc0:/dev/rtc0 --cap-drop ALL -it \
             --entrypoint /bin/sh sergioprado/alpine-app:1.0.0

# getpcaps 1
1: =

./app
Current RTC date/time is 2022-06-08 13:56:56

$ docker run --rm --device /dev/rtc0:/dev/rtc0 --cap-drop ALL \
             sergioprado/alpine-app:1.0.0
Current RTC date/time is 2022-06-08 14:05:48
```

RESTRICTING SYSCALLS

- A large number of system calls are exposed to every userspace process, and that means a larger surface attack.
  - If there is a bug in a system call, an exploited application could leverage the bug in the kernel to escalate privileges.
- Seccomp (Secure computing mode) provides a mechanism for a process to specify a filter for system calls, reducing the total kernel surface exposed to the application.
- By default, Docker runs containers with a default seccomp profile that currently disables around 44 system calls out of 300+.
- The default profile can be overridden via the *--security-opt seccomp=<new_profile>* option.

Embedded Labworks

HANDSON: CUSTOM SECCOMP PROFILE

```
{
  "defaultAction": "SCMP_ACT_ERRNO",
  "architectures": [
    "SCMP_ARCH_X86_64"
  ],
  "syscalls": [
    {
      "names": [
        "accept",
        "accept4",
        "access",
        "adjtimex",
        "alarm",
        "bind",
        "chdir",
        ...
```

```
1 $ docker run --rm --device /dev/rtc0:/dev/rtc0 \
2              --security-opt seccomp=seccomp-profile.json \
3              sergioprado/alpine-app-static:1.0.0
4 Current RTC date/time is 2022-06-08 18:35:09
5
6 $ docker run --rm --device /dev/rtc0:/dev/rtc0 \
7              --security-opt seccomp=seccomp-profile-noioctl.json \
8              sergioprado/alpine-app-static:1.0.0
9 RTC_RD_TIME ioctl: Operation not permitted
```

MANAGING RESOURCE USAGE

- Control Groups (cgroups) is a feature of the Linux kernel that allows to limit the access processes have to system resources such as CPU, RAM, block I/O and network.
- While the cgroups feature doesn't prevent privilege escalation, it is essential to prevent some denial-of-service (DoS) attacks.
- We should always limit the resources allocated to containers, especially those that act as servers.
- There are several options in Docker to configure resources allocated to containers.

CGROUPS OPTIONS IN DOCKER

Option	Description
--memory	Memory limit
--memory-swap	Total memory limit (memory + swap)
--cpu-shares	CPU shares (relative weight)
--cpus	Number of CPUs
--cpuset-cpus	CPUs in which to allow execution (0-3, 0,1)
--pids-limit	Tune container pids limit (set -1 for unlimited)
--device-read-iops	Limit read rate (IO per second) from a device
--device-write-iops	Limit write rate (IO per second) to a device

HANDS-ON: LIMITING CPU AND MEMORY

```
$ docker run --rm --device /dev/rtc0:/dev/rtc0 -it --entrypoint /bin/sh \
             sergioprado/alpine-app:1.0.0

$ docker stats
CONTAINER ID   NAME                CPU %     MEM USAGE / LIMIT     MEM %     NET I/O
07a2a3bbd59c   admiring_bhaskara   0.00%     1.266MiB / 31.29GiB   0.00%     7.56kB / 0B

$ docker run --rm --device /dev/rtc0:/dev/rtc0 --memory 512m --cpus 1 -it
             --entrypoint /bin/sh sergioprado/alpine-app:1.0.0

$ docker stats
CONTAINER ID   NAME        CPU %     MEM USAGE / LIMIT   MEM %     NET I/O       BLOCK I/O
d7ce4e892fa3   zen_kalam   0.00%     1.555MiB / 512MiB   0.30%     5.01kB / 0B   0B / 0B
```

LINUX SECURITY MODULES

- The Linux Security Module (LSM) framework provides a mechanism for various security checks to be hooked by kernel extensions.
- This framework is used to implement Mandatory Access Control (MAC) extensions such as SELinux, Smack, Tomoyo, and AppArmor.
- Docker is usually installed with AppArmor enabled by default, and a profile called *docker-default* is applied to new containers.
  - When running a container, the option *--security-opt apparmor=<new-profile>* can be used to change the default profile.
  - The option *--security-opt apparmor=unconfined* can be used to disable AppArmor.

HANDS-ON: CHECK IF APPARMOR IS ENABLED

```
$ docker info | grep Security -A 3
WARNING: No swap limit support
 Security Options:
  apparmor
  seccomp
   Profile: default

$ sudo apparmor_status
apparmor module is loaded.
68 profiles are loaded.
64 profiles are in enforce mode.
   /snap/core/13250/usr/lib/snapd/snap-confine
   /snap/core/13250/usr/lib/snapd/snap-confine//mount-namespace-capture-helper
   /snap/core/13308/usr/lib/snapd/snap-confine
   /snap/core/13308/usr/lib/snapd/snap-confine//mount-namespace-capture-helper
   /usr/bin/evince
   /usr/bin/evince-previewer
   /usr/bin/evince-previewer//sanitized_helper
   /usr/bin/evince-thumbnailer
   /usr/bin/evince//sanitized_helper
   ...
   docker-default
   ...
```

HANDS-ON: CUSTOM APPARMOR PROFILE

```
#include <tunables/global>

profile docker-app {
  #include <abstractions/base>

  /app ix,
  /dev/rtc0 r,

  deny /dev/[^rtc0]* rwklx,
  deny /proc/** rwklx,
  deny /sys/** rwklx,
}
```

```
$ sudo apparmor_parser --add docker-app

$ docker run --rm --device /dev/rtc0:/dev/rtc0 --security-opt apparmor=docker-app \
             sergioprado/alpine-app-static:1.0.0
Current RTC date/time is 2022-06-09 12:52:37
```

HANDS-ON: REMOVING ACCESS TO /DEV/RTC0

```
#include <tunables/global>

profile docker-app-nortc {
  #include <abstractions/base>

  /app ix,

  deny /dev/[^rtc0]* rwklx,
  deny /proc/** rwklx,
  deny /sys/** rwklx,
}
```

```
1 $ sudo apparmor_parser --add docker-app-nortc
2
3 $ docker run --rm --device /dev/rtc0:/dev/rtc0 --security-opt apparmor=docker-app-nortc \
4              sergioprado/alpine-app-static:1.0.0
5 /dev/rtc0: Permission denied
```

SECURING CONTAINER NETWORKING

- Don't use Docker's default bridge (*docker0*).
- When a container is created, Docker connects it to the *docker0* network by default.
- Therefore, all containers are connected to *docker0* and can communicate with each other.
- Instead, create a custom network with the *docker network* command, and use it to start containers.

```
$ docker network create app-network
$ docker run --network app-network ...
```

SECURING CONTAINER NETWORKING (CONT.)

- Avoid sharing the host's network namespace (*--network=host*).
  - A TCP port in the container's network can be mapped to the host via the *--publish* parameter.
- Don't expose the Docker daemon socket inside a container (*/var/run/docker.sock*).
- Use TLS to secure communication between services running inside containers.

SECURING CONTAINER STORAGE

- Mount the container's root filesystem as read-only using the *--read-only* option (remember, a container should be immutable):

```
$ docker run --read-only ...
```

- If needed, mount a temporary filesystem to store non-persistent data via the *--tmpfs* or *--mount* options:

```
$ docker run --tmpfs /run:rw,noexec,nosuid,size=65536k ...
```

SECURING CONTAINER STORAGE (CONT.)

- Docker provides two options for persistent data storage:
  - Volume: independent and filesystem-agnostic storage that can only be accessed inside containers.
  - Bind mount: storage is just a directory mounted inside the container, also visible to the host OS.
- In case you are storing sensitive information, you might want to consider encryption.
- Docker has a command called *secret* to help manage the storage and retrieval of secrets (usernames/passwords, TLS certificates and keys, SSH keys, etc).

CONCLUSION: DEFENSE IN DEPTH

- The first command used to run the container works, but as we could see during the presentation, it's very insecure!

```
$ docker run --rm -v /dev:/dev --privileged sergioprado/alpine-app-static:1.0.0
```

- After all the mitigation techniques learned during this presentation, we can come up with the following command, that also works, and it's much more secure!

```
$ docker run --rm \
          --device /dev/rtc0:/dev/rtc0 \
          --read-only \
          --tmpfs /run:rw,noexec,nosuid,size=65536k \
          --cap-drop all \
          --security-opt no-new-privileges \
          --security-opt seccomp=seccomp-profile.json \
          --security-opt apparmor=docker-app \
          --network app-network \
          --memory 512m \
          --cpus 1 \
          sergioprado/alpine-app-static:1.0.0
```

EMBEDDED LINUX CONFERENCE 2022

THANK YOU! QUESTIONS?

Sergio Prado, Embedded Labworks
sergio.prado@e-labworks.com
https://www.linkedin.com/in/sprado
https://twitter.com/sergioprado